01/08/25

Searching

Searching is the most common operation performed by a database system

Linear Search:

- Baseline for efficiency
- Start at the beginning of a list and proceed element by element until:
    - You find what you're looking for
    - You get to the last element and haven't found it

Record: a collection of values attributes of a single entity instance; a row of a table

Lists of Records

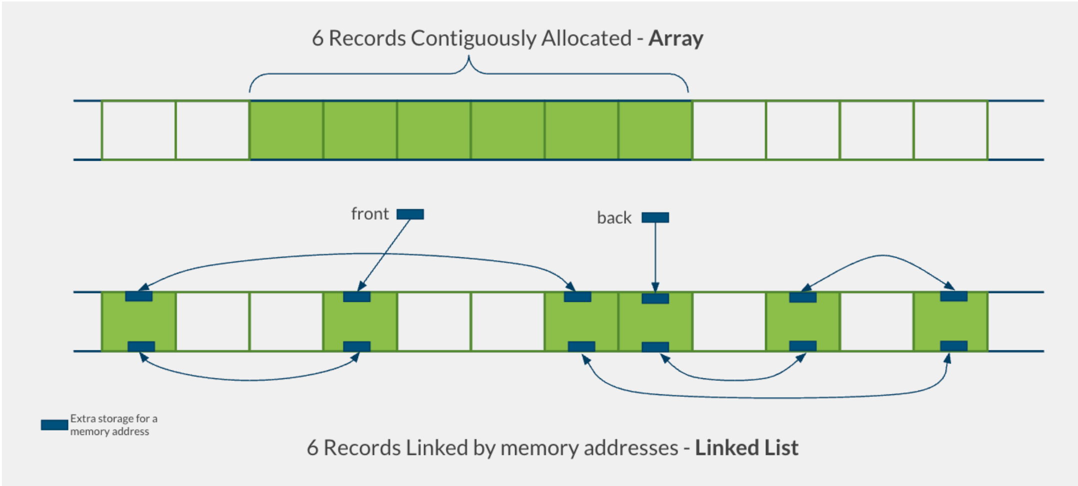
If each record takes up x bytes of memory, then for n records, we need n * x bytes of memory

Contiguously allocated list: all n * x bytes are allocated as a single "chunk" of memory

- Arrays

Linked list: each record needs x bytes + additional space for 1 or 2 memory addresses

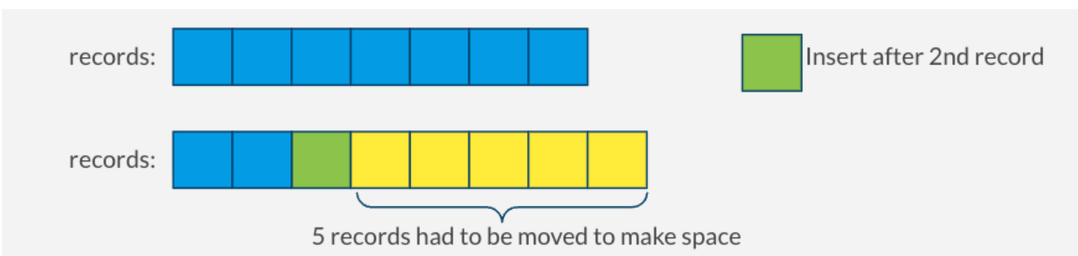
- Individual records are linked together in a type of chain using memory addresses



Pros and Cons

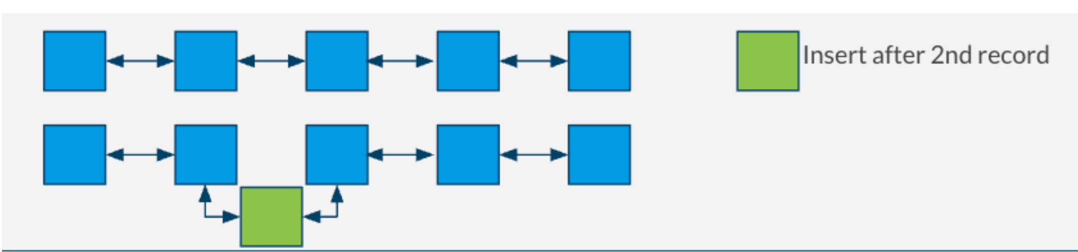
Arrays

- Faster for random access
- Slow for inserting anywhere but the end



Linked Lists

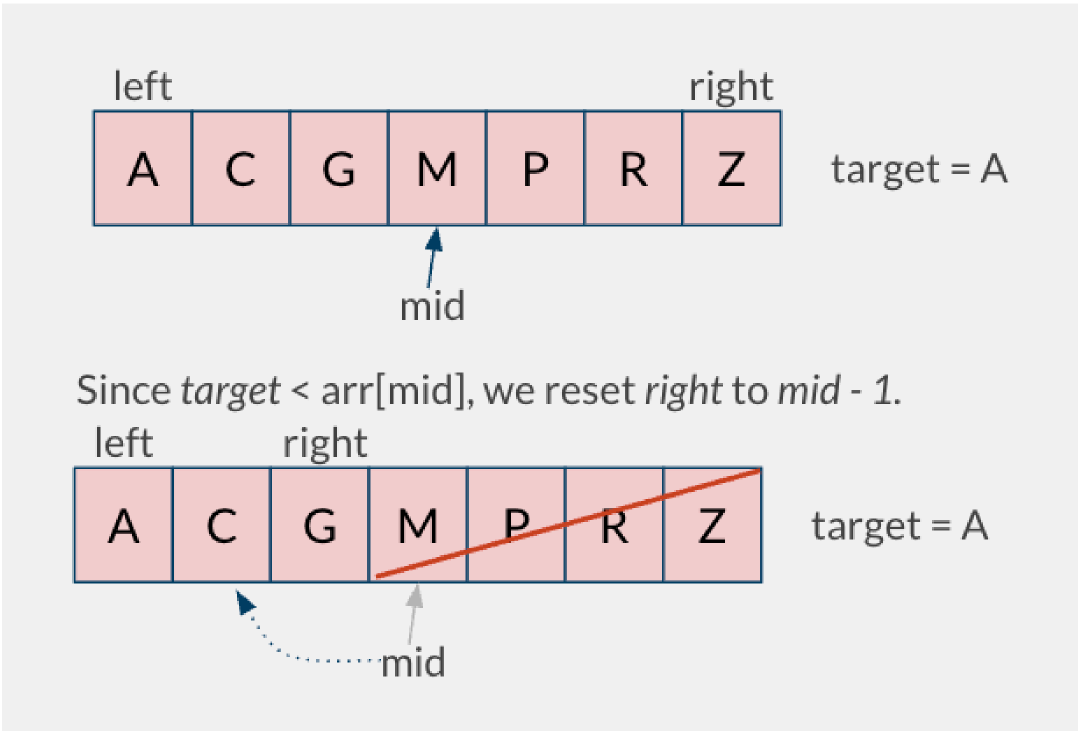
- Faster for inserting anywhere in the list
- Slower for random access



Binary Search

Input: array of values in sorted order, target value

Output: the location (index) of where target is located or some value indicating target was not found



Time Complexity

Linear Search

- Worst case: O(n)

Binary Search

- Worst case: O($log_2 n$)

Database Searching

Assume data is stored on disk by column id's value

- Searching for a specific id is fast

But what if we want to search for a specific specialVal?

- Only option is linear scan of that column

Can’t store data on disk sorted by both id and specialVal (at the same time)

- Data would have to be duplicated → space inefficient

id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

What data structure could we use?

- Array of tuples
    - Same issue as above
- Linked list of tuples
    - Same issue as above

Something with fast insert and fast search?

- Binary Search Tree: a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent

Binary Search Trees

Assuming that the keys of a BST are pairwise distinct

Each node has the following attributes:

- *p*, *left*, and *right*, which are pointers to the parent, the left child, and the right child, respectively
- *key*, which is key stored at the node

Traversal of the Nodes in a BST

Traversal: visiting all the nodes in a graph

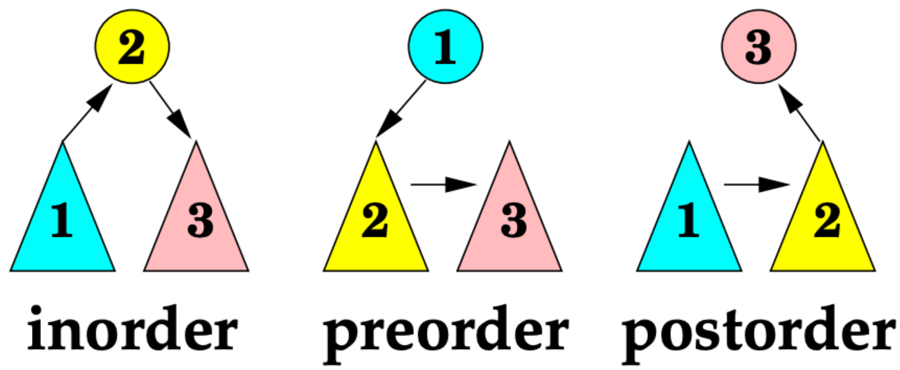
- Strategies are specified by the ordering of the three objects to visit: the current node, the left subtree, and the right subtree
- Assume the left subtree always comes before the right subtree

Three strategies

- Inorder: left subtree, current node, right subtree
- Preorder: current node, left subtree, right subtree
- Postorder: left subtree, right subtree, current node

Inorder Travel Pseudocode

```
Inorder-Walk(x)
   1: if x = nil then return
   2: Inorder-Walk(left[x])
   3: Print key[x]
   4: Inorder-Walk(right[x])
```



Inorder travel on a BST finds the keys in nondecreasing order

Operations on BST

Searching for a key

Assuming that a key and the subtree in which the key is searched for are given as an input

Suppose we are at a node

- If the node has the key that is being searched for, then the search is over
- Otherwise, the key (of the current node) is either strictly smaller than the key that is searched for or strictly greater than the key that is searched for
  - If the current key is smaller than the key that is searched for, then the key that is searched for must be in the right subtree (by the properties of BST)
  - If the current key is larger than the key that is searched for, then the key that is searched for must be in the left subtree

```
// k is the key that is searched for, and x is the start node
BST-Search(x, k)
  1: y ← x
  2: while y != nil do
    3: if key[y] = k then return y
```

```
    4: else if key[y] < k then y ← right[y]
    5: else y ← left[y]
  6: return ("NOT FOUND")
```

Maximum and minimum

Minimum: identify the leftmost node

- Find the farthest node you can reach by following only left branches

```
BST-Minimum(x)
  1: if x = nil then return ("Empty Tree")
  2: y ← x
  3: while left[y] != nil do y ← left[y]
  4: return (key[y])
```

Maximum: identify the rightmost node

- Find the farthest node you can reach by following only right branches

```
BST-Maximum(x)
  1: if x = nil then return ("Empty Tree")
  2: y ← x
  3: while right[y] != nil do y ← right[y]
  4: return (key[y])
```

Insertion

Suppose that we need to insert a node z such that k = key[z]

Using binary search we find a nil such that replacing it by z does not break the BST-property

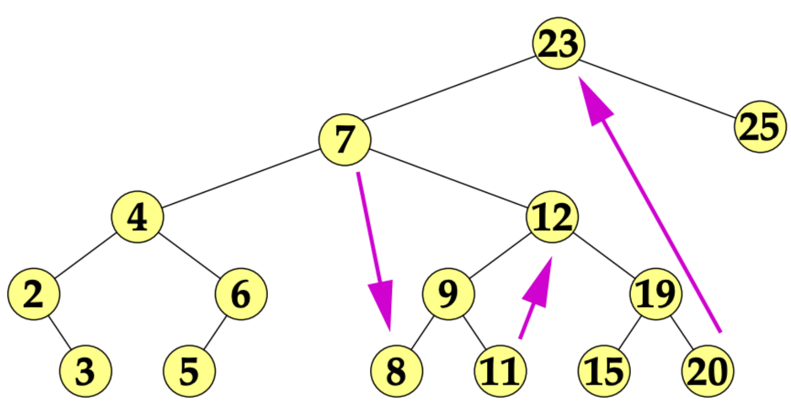
```
BST-Insert(x, z, k)
  1: if x = nil then return "Error"    // check if root is null
  2: y ← x                             // assign y to x
  3: while true do {                   // will loop until broken
    4: if key[y] < k                   // if current key is less than insert key
```

```
    5: then z ← left[y]              // assign z to left subnode
    6: else z ← right[y]             // otherwise, assign z to right subnode
    7: if z = nil break              // if z is now null (subnode doesn't exist, break the
    7.5: y ← z                       // not in the notes, but I think it's needed to advance
  8: }
  9: if key[y] > k then left[y] ← z    // if current key is less than insert key, assign
  10: else right[p[y]] ← z            // otherwise, assign right subnode to z
```

Successor and Predecessor

Successor: for a key k in a BST, the smallest key that belongs to the tree and that is strictly greater than k

- If node x has the right child, then the successor is the minimum in the right subtree of x
- Otherwise, the successor is the **parent** of the farthest node that can be reached from x by following only right branches backward




```
BST-Successor(x)
  1: if right[x] != nil then
  2: { y ← right[x]
    3: while left[y] != nil do y ← left[y]
    4: return (y) }
  5: else
```

```
6: { y ← x
   7: while right[p[x]] = x do y ← p[x]
   // x should get reassigned to y in the while loop
   8: if p[x] != nil then return (p[x])
   9: else return ("NO SUCCESSOR") }
```

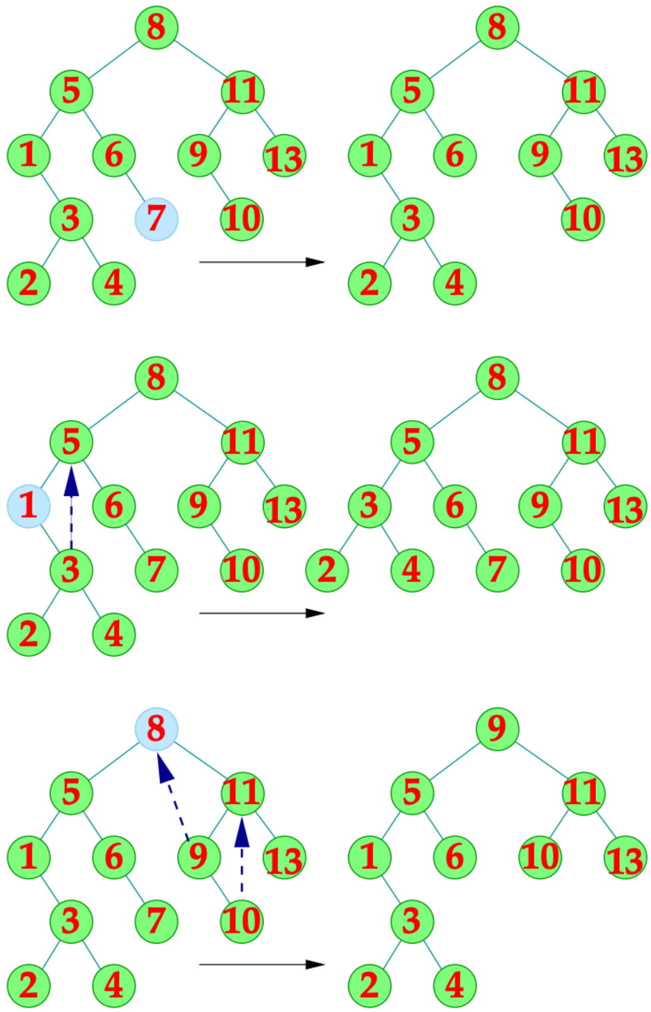
Predecessor: for a key k in a BST, the largest key that belongs to the tree and that is strictly less than k

- If node x has the left child, then the successor is the maximum in the left subtree of x
- Otherwise, the successor is the **parent** of the farthest node that can be reached from x by following only left branches backward

Deletion

Suppose we want to delete a node z

1. If z has no children, then we will just replace z by nil
2. If z has only one child, then we will promote the unique child to z's place
3. If z has two children, then we will identify z's successor. Call it y. The successor y either is a leaf or has only the right child (because it is a successor). Promote y to z's place. Treat the loss of y using one of the above two solutions.




```
// This algorithm deletes z from BST T
BST-Delete(T, z)
   if left[z] = nil or right[z] = nil
   then y ← z
   else y ← BST-Successor(z)
✄ y is the node that’s actually removed.
✄ Here y does not have two children.
   if left[y] != nil
   then x ← left[y]
   else x ← right[y]
✄ x is the node that’s moving to y’s position.
```

```
    10: if x != nil then p[x] ← p[y]
11: ✄ p[x] is reset If x isn't NIL.
12: ✄ Resetting is unnecessary if x is NIL.
    13: if p[y] = nil then root[T] ← x
14: ✄ If y is the root, then x becomes the root.
15: ✄ Otherwise, do the following.
    16: else if y = left[p[y]]
        17: then left[p[y]] ← x
18: ✄ If y is the left child of its parent, then
19: ✄ Set the parent's left child to x.
    20: else right[p[y]] ← x
21: ✄ If y is the right child of its parent, then
22: ✄ Set the parent's right child to x.
    23: if y != z then
    24: { key[z] ← key[y]
        25: Move other data from y to z }
27: return (y)
```